Fifth IEEE Workshop on Future Trends of Distributed Computing Systems, FTDCS '97,
Tunisia, October 1997.

Analysing the Behaviour of Distributed Software Architectures: a Case Study

Jeff Magee, Jeff Kramer and Dimitra Giannakopoulou

{jnm,dg1,jk}@doc.ic.ac.uk
Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK.

Abstract

A Software Architecture is the overall structure of a system in terms of its constituent components and their interconnections. In this paper, we describe work to associate behavioural specifications with the components of a distributed software architecture and an approach to analysing the behaviour of systems composed from these components. The approach is based on the use of Labelled Transition Systems to specify behaviour and Compositional Reachability Analysis to check composite system models. The architecture description of a system is used directly to generate the model used for analysis. Analysis allows a designer to check whether an architecture satisfies the properties required of it. The approach is illustrated using a case study of an Active Badge system.

1. Introduction

Software Architecture has been identified as a promising approach to bridging the gap between requirements and implementations in the design of complex systems. A Software Architecture describes the gross organisation of a system in terms of its components and their interactions. The initial emphasis in Software Architecture specification has thus been in capturing system structure[5]. The authors have previously published papers on the use of the architecture description language Darwin for specifying the structure of distributed systems and subsequently directing the construction of those systems[8,9]. Darwin can be used to organise CORBA based distributed systems[10]. Darwin describes a system in terms of components which manage the implementation of services.. Structure is specified in Darwin by declaring the bindings between the services required and provided by component instances. Darwin has both graphical and textual forms with appropriate tool support.

In this paper we investigate the use of Darwin as the framework for specifying the component structure and their potential interactions for the purpose of behaviour analysis rather than system construction. Darwin has been designed to be sufficiently abstract to support multiple views (cf. [7]), two of which are the behavioural view (for behaviour analysis) and the service view (for construction) (Figure 1). Each view is an elaboration of the basic structural view: the skeleton upon which we hang the flesh of behaviour specification or service implementation.

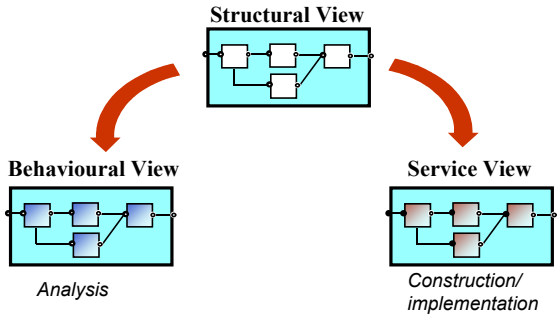


Figure 1. Common Structural View with Service and Behavioural Views

In previous papers we have discussed the use of Darwin to produce the service view, with components providing and requiring services at their interfaces and with implementation definitions for the primitive components. In this paper we concentrate on the behavioural view using labelled transition systems (LTS) for behaviour specification and analysis. We have chosen to use LTS as it supports the appropriate compositionality (using Compositional Reachability Analysis CRA) with the components specified simply as finite state automata[3]. In addition, we have techniques for analysing for both safety [2] and liveness [3] properties. This is supported by software tools which provide for automatic composition, analysis, minimisation, animation and graphical display. Rather than describe the underlying theory or details of the analysis techniques and their relation to Darwin, in this paper, we illustrate the approach by example.

2. Architecture Behaviour specification: an Example

The example is taken from an Active Badge personnel location system implemented by one of the authors in the Regis programming environment using hardware from Olivetti Research Laboratories in Cambridge. A description of this system together with a description of the implementation architecture in Darwin may be found in [9]. Active Badges emit and receive infrared signals which are received/transmitted by a network of infrared sensors connected to workstations as depicted in Figure 2. The system permits the location and paging of badge wearers within a building.

2.1 Sensor network

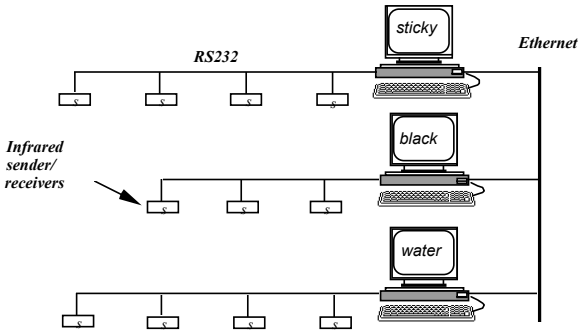
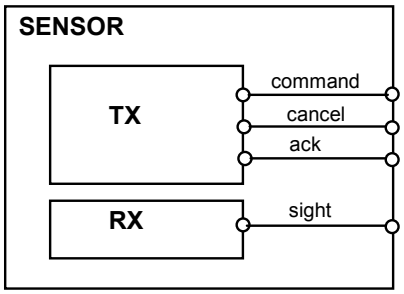


Figure 2 - Active Badge Sensor Network

Each sensor consists of transmitter and receiver sub-components. The receiver produces badge sighting indications at its interface while the transmitter accepts commands to page a specific badge. Commands may consist of directions to play a tune on the badge's speaker or to turn on an LED. In this analysis, we do not distinguish between the various forms of command. Commands may be cancelled before they have been received by a badge or they may be acknowledged if sent successfully. The structure of a *SENSOR* component is depicted in Figure 3 together with the Darwin description of that structure.

The Darwin description of *SENSOR* declares an instance of each of the primitive component types *RX* and *TX* representing the receive and transmit behaviour respectively. The interface of the component consists of a set of primitive actions which are bound to their counterparts in the primitive component specification. In an implementation these actions, which are declared as *portals* here, become services either provided or required by a component[8]. In this more abstract view, no commitment is made as to the location of the implementation of an action, calling direction and datatype




```
component SENSOR {
   portal command; cancel; ack; sight;
   inst TX; RX;
   bind
      command -- TX.command;
      cancel  -- TX.cancel;
      ack     -- TX.ack;
      sight   -- RX.sight;
}
```

Figure 3 - Description of SENSOR component in Darwin

Primitive Component behaviour

The behavioural specification for *SENSOR* involves describing each of its primitive sub-components as a finite state automata in a CSP-like[6] notation. This notation is used as a concise way of describing the Labelled Transition System (LTS) of the component for analysis purposes. It is an "ascii" notation to simplify parsing by the analysis tools.

```
const NBAD = 2  //number of badges

RX = (sight[b:1..NBAD] -> RX).
```


```
TX = (command[b:1..NBAD] ->
        (ack[b] -> TX
         | cancel[b] -> TX
         | command[c:1..NBAD]->ERROR)).
```

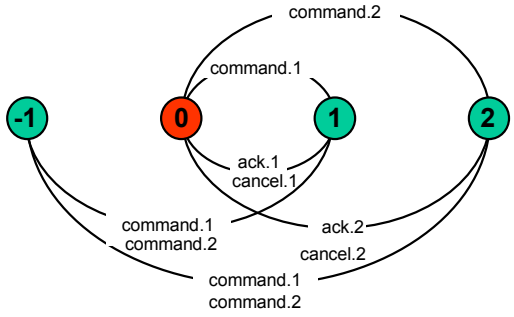


Figure 4 - Component RX & TX behaviour specification (transitions go clockwise, -1 labels the ERROR state).

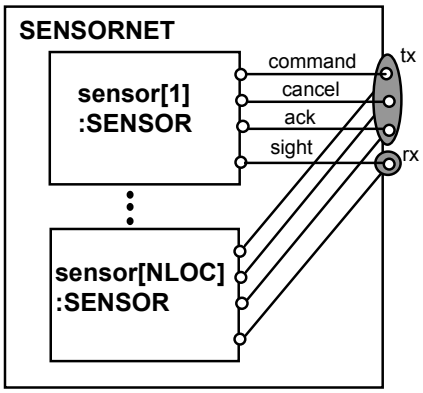
Figure 4 gives the primitive component specification together with its LTS in diagrammatic form for *RX* and *TX*. These diagrams are generated by our analysis tools as an aid to comprehension. The diagrams here illustrate the model for two badges (ie *NBAD = 2*). The *RX* specification simply states that a badge sighting for a badge can occur at any time. The specification for *TX* states that a command for a specific badge may be followed either by a request to cancel that command or an acknowledgement from that badge. The sensor hardware can only buffer one command to be sent to a badge at a time. Consequently, the specification states that an attempt to carry out another command before the previous command is either cancelled or acknowledged is an error. Note that actions are indexed/subscripted by the identity of the badge they apply to. In the implemented badge system, the badge identity is a parameter to actions. In general, action subscripts provide a convenient way to model parameters for the purposes of analysis, although care must be taken to restrict the range of subscripts or intractable models may result.

Composite Components

Composite components are simply the parallel composition of their constituent behaviours. Processes synchronise on shared actions. For example, the Darwin specification for the SENSOR component generates the following LTS specification:

```
||SENSOR =(RX || TX)
          @ {command, cancel, ack, sight}.
```

The LTS notation distinguishes between process expressions which may use action prefix (`->`), choice (`|`) and recursion to specify primitive component behaviour and composition expressions which may only use parallel composition (`||`) and re-labelling (`/`). These restrictions mean that the notation can only describe finite and, consequently, potentially analysable models. The simple composition expression for *SENSOR* does not require re-labelling since actions are referred to by the same label in the composite component as in the constituent primitive components. As we will see in the following, in general, bind statements in Darwin lead to re-labelling in the LTS specification. The @ symbol specifies the set of action labels (alphabet) which are visible at the interface of the component and thus may be shared with other components. It restricts the alphabet of the LTS to the actions which are prefixed by these labels. All other actions are "hidden" and will appear as "silent" or "tau" actions during analysis if they do not disappear during minimisation. Figure 5 gives the composite component structure for *SENSORNET*.




```
interface TRANS {
   command[1..NBAD][1..NLOC];
   cancel [1..NBAD][1..NLOC];
   ack    [1..NBAD][1..NLOC];
}

interface REC  {
   sight  [1..NBAD][1..NLOC];
}

component SENSORNET {
   portal
      tx: TRANS;
      rx: REC;

   forall j = 1 to NLOC {
      inst sensor[j]:SENSOR;
      forall k = 1 to NBAD bind
         sensor[j].command[k]
                  -- tx.command[k][j];
         sensor[j].cancel[k]
                  -- tx.cancel[k][j];
         sensor[j].ack[k]
                  -- tx.ack[k][j];
         sensor[j].sight[k]
                  -- rx.sight[k][j];
   }
}
```

Figure 5 - composite component SENSORNET .

A separate instance of *SENSOR* is declared for each location at which a badge may be detected. There are *NLOC* locations numbered from 1 to *NLOC*. The interface to *SENSORNET* is structured using Darwin interface types *TRANS* and *REC*. In an implementation, these translate to IDL descriptions with subscripts becoming operation parameters. In the behavioural specifications, interface instance names simply become action label prefix names. Interested readers who consult the badge system implementation in [3] will see that multiplex and de-multiplex components are required to implement the complex binding between sensor components and the interface to the *SENSORNET* component. This is because in our abstract description, actions and thus interfaces do not have a physical location whereas in general an

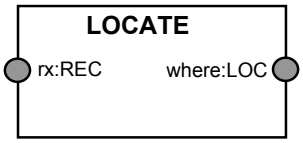
implementation will fix the provision of an interface to a specific server. In the sensor network, as depicted in Figure 2, sensors are controlled by different machines. The behavioural specification which corresponds to the Darwin description of *SENSORNET* is given below:

```
||SENSORNET = (sensor[1..NLOC]:SENSOR)
   /{rx.sight[i:1..NBAD][j:1..NLOC]
      /sensor[j].sight[i],
     tx.command[i:1..NBAD][j:1..NLOC]
      /sensor[j].command[i],
     tx.ack[i:1..NBAD][j:1..NLOC]
      /sensor[j].ack[i],
     tx.cancel[i:1..NBAD][j:1..NLOC]
     /sensor[j].cancel[i]}
      @{rx,tx}.
```

Component instantiation is modelled by creating a copy of a process (eg. *SENSOR*) with each action label contained in the process prefixed by the instance name. Thus the `command[1]` action provided by *SENSOR* becomes `sensor[2].command[1]` in the instance `sensor[2].` Re-label specifications are of the form new-label/old-label.

2.2 Location Service

The badge system location service stores the current location of each badge as determined from the sighting events generated by the sensor network. The location service is represented as the component *LOCATE* with interfaces as described below in Figure 6. The interface of type *REC* is as described in Figure 5. The *LOC* interface provides a means to *query* the location of a badge with the location of that badge reported by a corresponding *at* event. If a badge changes location then the location service uses the *at* event to signal the change spontaneously .




```
interface LOC {
   query[1..NBAD];  at[1..NBAD][1..NLOC];
}
```

Figure 6 - location service component SENSORNET

The behaviour of *LOCATE* was implemented as a single process in Regis, however, to specify the behaviour succinctly, we can utilise concurrency. The current location of each badge is stored by a BADGELOC process. The LOCATE service is then the concurrent composition of these processes as shown below.

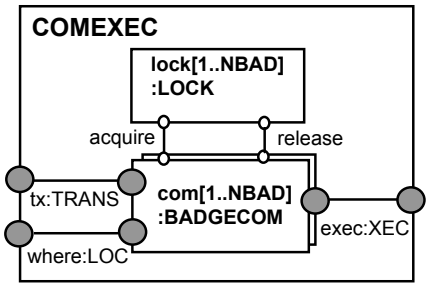
```
BADGELOC = BADGELOC[1],
BADGELOC[i:1..NLOC]
   =(sight[i] -> BADGELOC[i]
     |{sight[j:1..i-1],sight[j:i+1..NLOC]}
        -> at[j] -> BADGELOC[j]
      | query -> at[i] -> BADGELOC[i]
      ).

// service stores locations of all badges
||LOCATE = (badgeloc[1..NBAD]:BADGELOC)
     /{rx.sight[i:1..NBAD][j:1..NLOC]
       /badgeloc[i].sight[j],
      where.query[i:1..NBAD]
       /badgeloc[i].query,
      where.at[i:1..NBAD][j:1..NLOC]
       /badgeloc[i].at[j]}
       @{rx,where}.
```

It would be possible to describe *LOCATE* as a composition of *BADGELOC* processes using Darwin, however, the concurrency in *LOCATE* is more an artefact of the modelling technique than a requirement of the architecture, consequently, this was not done. This has not restricted the freedom to have *LOCATE* as either a primitive or a composite component during design and implementation.

2.3 Command Execution

Executing a badge command is analogous to setting up a telephone call. As with telephone call processing software, the Regis implementation of the badge system created a new component/ process to deal with each request to execute a badge command[9]. When describing behaviour, we can only accommodate static process structures and consequently, command execution is modelled by replication with one *BADGECOM* per badge. When requested to page a badge, the BADGECOM component must first determine the location of that badge and then reserve the sensor at that location for its exclusive use. This exclusion is required since, as specified in section 2.1, each sensor has only a single command buffer. This exclusion is provided by a LOCK component per location. The resulting structure is depicted in Figure 7.



```
interface  XEC {
   page[1..NBAD]; confirm[1..NBAD]; }
```

Figure 7 - command execution component COMEXEC

In the interests of brevity, we have omitted the Darwin description. The behavioural specification of the LOCK component is shown below:

```
LOCK =(acquire[i:1..NBAD]
       -> release[i] -> LOCK).
```

This specifies that when the lock is acquired by a *BADGECOM* component it must be released by the same component (same index) before the lock can be acquired again. The specification for *BADGECOM* is:

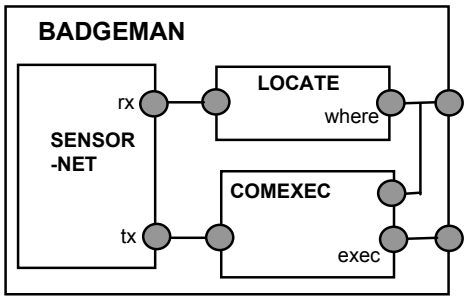
```
BADGECOM
     =(page-> QUERY
       |at[j:1..NLOC] -> BADGECOM),
QUERY
     =(query -> at[i:1..NLOC] -> GETLOCK[i]
       |at[j:1..NLOC] -> GETLOCK[j]),
GETLOCK[i:1..NLOC]
     =(acquire[i] -> EXEC[i]
       |at[j:1..NLOC] -> GETLOCK[j),
EXEC[i:1..NLOC]
     =(command[i] -> WAITACK[i]
       |at[j:1..NLOC] -> release[i]
                        ->GETLOCK[j]),
WAITACK[i:1..NLOC]
      =(ack[i] -> release[i] -> success
                              -> BADGECOM
       |at[j:1..NLOC] -> cancel[i]
        -> release[i] -> GETLOCK[j]).
```

Note that at any stage of the command execution protocol, it is possible for the badge to change location (signalled by *at*) and consequently, BADGECOM must release the lock for that location and cancel the command before trying to execute the command at the new location(see *EXEC*). For completeness, the specification of COMEXEC is:

```
||COMEXEC = (com[1..NBAD]:BADGECOM
             || lock[1..NLOC]:LOCK)
       /{lock[i:1..NLOC].acquire[j:1..NBAD]
          /com[j].acquire[i],
         lock[i:1..NLOC].release[j:1..NBAD]
          /com[j].release[i],
         tx.command[i:1..NBAD][j:1..NLOC]
         /com[i].command[j],
         tx.cancel[i:1..NBAD][j:1..NLOC]
         /com[i].cancel[j],
         tx.ack[i:1..NBAD][j:1..NLOC]
         /com[i].ack[j],
         where.query[i:1..NBAD]
         /com[i].query,
         where.at[i:1..NBAD][j:1..NLOC]
         /com[i].at[j],
         exec.page[i:1..NBAD]
         /com[i].page,
         exec.confirm[i:1..NBAD]
         /com[i].success
                  } @{where,exec,tx}.
```

2.4 Active Badge service

The complete active badge service is provided by the *BADGEMAN* component as depicted in Figure 8 together with the composition expression describing its behaviour. The interested reader may again check with [9] to see that the architecture of the original implementation architecture has been closely followed in developing the behaviour specification.




```
||BADGEMAN
    = (SENSORNET || LOCATE || COMEXEC)
      @ {where, exec}
```

Figure 8 - Active Badge Service

3. Architecture Behaviour Analysis

The question that now must be answered is; given the behaviour specification, what can we do with it? The approach we have taken is to use compositional reachability analysis to perform an exhaustive search of the state space of the LTS model generated from the behaviour specification.

Deadlock

The reachable state space for a system with two badges and five locations consists of 202,275 states (871,350 transitions). In the example outlined in the foregoing, there are no reachable deadlock or error states. However, an earlier version of the specification which omitted the line from BADGECOM shown in italics below:

```
BADGECOM =(page-> QUERY
           |at[j:1..NLOC] -> BADGECOM),
QUERY    =(query -> at[i:1..NLOC]
 -> GETLOCK[i]
          |at[j:1..NLOC] -> GETLOCK[i]),
```

This did result in a deadlock. Our analysis tools produced the following trace of an action sequence in a system with two badges (NBAD=2) which would lead to deadlock:

```
Trace to DEADLOCK:
   <rx.sight.1.2,  rx.sight.2.2
    exec.page.1, exec.page.2>
```

This is the situation where each badge has moved to location 2 and consequently the *LOCATE* component is trying to execute the actions indicating there has been a change of location (`at.1.2, at.2.2`). However, users have instigated page commands on both badges and consequently, the *BADGECOM* for each process is trying to execute a *query* on *LOCATE* and is thus not able to synchronise with *at*. Similarly, *LOCATE* cannot synchronise with *query* - thus deadlock.

Error States

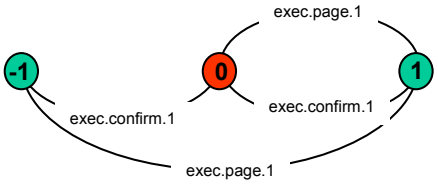
A (deliberately) incorrect implementation of the LOCK component which did not provide mutual exclusion yielded the following output from a reachability analysis:

```
Trace to property violation in SENSORNET:
  <rx.sight.1.2, rx.sight.2.2,
   exec.page.1,  where.at.1.2,
   tx.command.1.2, exec.page.2,
   where.at.2.2, tx.command.2.2>
```

This is clearly the case where two commands have been executed on the sensor at location 2.

Property Automata

Checks can be made that the model satisfies certain safety properties by specifying these properties as automata and composing them with the system[ ]. For example, the following property depicted in Figure 9 together with the automata it generates asserts that a page request for badge ID must always be followed by a confirmation.




```
property
 CONFIRM(ID=1) =(exec.page[ID]
                -> exec.confirm[ID]
                   -> CONFIRM).
```

Figure 9 - property CONFIRM

Test Scenarios

In addition to exhaustive testing, the analysis tools permit a user to animate various test scenarios and examine the resulting trace of actions. For example, the following trace is the scenario in which badge 1 changes location from 1 to 3 while there is a page request outstanding for it:

```
<exec.page.1, where.query.1
 where.at.1.1      -- badge 1 at location 1
 lock.1.acquire.1, tx.command.1.1
 rx.sight.1.3      -- badge 1 moved to location 3
 where.at.1.3, tx.cancel.1.1
 lock.1.release.1, lock.3.acquire.1
 tx.command.1.3, tx.ack.1.3
 lock.3.release.1, exec.confirm.1>
```

4. Conclusion

Software Architecture provides a sound basis for both design and implementation [8,9,10]. In this paper, we have illustrated the use of the Software Architectural specification of a system as the basis for analysis. Our approach to specifying the behaviour of architectural elements using a process calculus notation differs only in detail from that outlined in [1]. Our approach is distinguished by the direct use of the structural architecture description for both construction and analysis and in the analysis techniques utilised. We are currently investigating the applicability of our approach in the context of industrial case studies.

Acknowledgements

We gratefully acknowledge the EPSRC (Grant Ref: SAA GR/J52693 and TRACTA GR/J 87022) and the EU (ARES Framework IV contract 20477) for their financial support.

References

1. Allen R. and Garlan D., *Formalizing Architectural Connection*, (Proc. of 16th Int. Conf. on Software Engineering (ICSE 16), Sorrento, May 1994, 71-80.
2. Cheung S.C. and Kramer J., *Checking Subsystem Safety Properties in Compositional Reachability Analysis*, (Proc. of 18th IEEE Int. Conf. on Software Engineering (ICSE-18), Berlin, 1996), 144-154.
3. Cheung S.C., Giannakopoulou D., and Kramer J., *Verification of Liveness Properties using Compositional Reachability Analysis*, accepted for (ESEC/FSE 97), Zurich, Sept. 1997).
4. Giannakopoulou D., Kramer J. and Cheung S.C., *TRACTA: An Environment for Analysing the Behaviour of Distributed Systems*, (Proc. of 1st ACM SIGPLAN Workshop on Automatic Analysis of Software (AAS '97)), Paris, January 1997, 113-126.
5. Garlan D. and Perry D.E., *Introduction to the Special Issue on Software Architecture*, IEEE Transactions on Software Engineering, 21 (4), April 1995, pp 269-274.
6. Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
7. Kruchten P.B., The 4+1 Model of Architecture, IEEE Software, 12 (6), Nov. 1995, pp 42-50.
8. Magee J., Dulay N., Eisenbach S., Kramer J., *Specifying Distributed Software Architectures*, ESEC '95, Sitges, September 1995, LNCS 989, 1995, 137-153.
9. Magee J., Dulay N. and Kramer J., *Regis: A Constructive Development Environment for Distributed Programs*, Distributed Systems Engineering Journal, 1 (5), Special Issue on Configurable Distributed Systems, (1994), 304-312.
10. Magee J., Tseng A., Kramer J., *Composing Distributed Objects in CORBA*, (Third International Symposium on

Autonomous Decentralized Systems (ISADS 97), Berlin, Germany, April 9 - 11, 1997.